17장. 제네릭

17.1. 제네릭의 이해

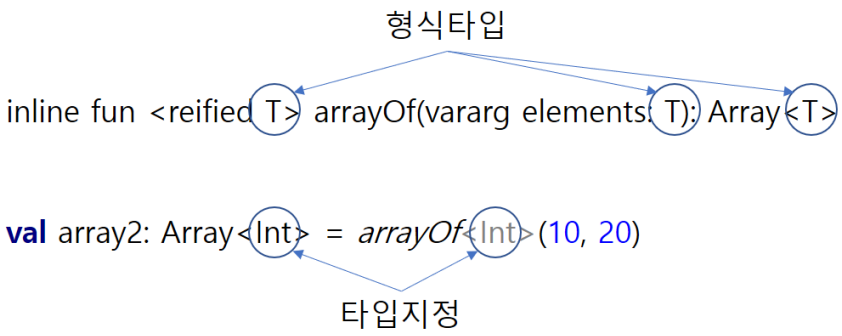
17.1.1. 제네릭이란?

- 제네릭이란 형식타입
- 타입을 예측할수 없거나 하나의 타입으로 고정할수 없는 경우
- 제네릭으로 형식 타입을 선언하고 실제 이용시 정확한 타입을 부여

```
val array = arrayOf("kkang", 10, true)
```

- inline fun <reified T> arrayOf(vararg elements: T): Array<T>

```
val array2: Array<Int> = arrayOf<Int>(10, 20)
```



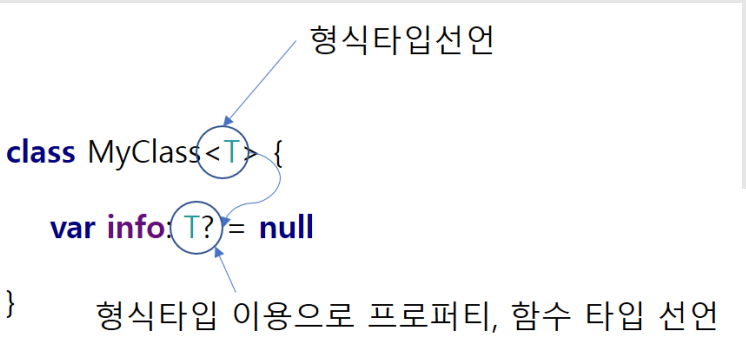
17.1. 제네릭의 이해

17.1.2. 제네릭 선언 및 이용

```
class MyClass {
    var info: String? = null
}
```

```
class MyClass {
    var info: T? = null//error
}
```

```
class MyClass<T> {
    var info: T? = null
}

fun main(args: Array<String>) {
    val obj1=MyClass<String>()
    obj1.info="hello"

    val obj2=MyClass<Int>()
    obj2.info=10
}
```



17.1. 제네릭의 이해

타입 유추에 의한 이용

```
class MyClass2<T>(no: T){
    var info: T? = null
}

fun main(args: Array<String>) {

    val obj3=MyClass2<Int>(10)
    obj3.info=20

    val obj4=MyClass2("hello")
    obj4.info="world"
}
```

형식타입 여러 개 선언

```
class MyClass<T, A> {
    var info: T? = null
    var data: A? = null
}

fun main(args: Array<String>) {
    val obj: MyClass<String, Int> = MyClass()
    obj.info="hello"
    obj.data=10
}
```

17.1. 제네릭의 이해

함수와 제네릭

```
class MyClass<T, A> {
    var info: T? = null
    var data: A? = null

    fun myFun(arg: T): A? {
        return data
    }
}
```

```
fun <T> someFun(arg: T): T? {
    return null
}
```

17.2. 제네릭 제약

17.2.1. 타입제약

- 제네릭 제약(Generic Constraint) 란 형식타입을 선언하면서 특정 타입만 대입되도록 제약하는 것

```
class MathUtil<T: Number> {
    fun plus(arg1: T, arg2: T): Double {
        return arg1.toDouble() + arg2.toDouble()
    }
}

fun main(args: Array<String>) {
    val obj = MathUtil<Int>()
    obj.plus(10, 20)

    val obj2 = MathUtil<Double>()

//    val obj3 = MathUtil<String>()//error
}
```

17.2. 제네릭 제약

여러 개의 타입으로 제약

```
interface MyInterface1
interface MyInterface2

class MyHandler1: MyInterface1, MyInterface2

class MyHandler2: MyInterface1

class  MyClass<T> where T: MyInterface1, T: MyInterface2{
    //.............
}

fun main(args: Array<String>) {
    val obj = MyClass<MyHandler1>()

    val obj2 = MyClass<MyHandler2>()//error
}
```

17.2. 제네릭 제약

17.2.2. Null 불허 제약

- 제네릭의 형식타입은 기본으로 Nullable로 선언

```
class MyClass<T> {
    fun myFun(arg1: T, arg2: T){
        println(arg1?.equals(arg2))
    }
}

fun main(args: Array<String>) {
    val obj = MyClass<String>()
    obj.myFun("hello", "hello")

    val obj2 = MyClass<Int?>()
    obj2.myFun(null, 10)
}
```

17.2. 제네릭 제약

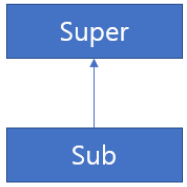
- Non-Nullable 로 선언

```
class MyClass<T: Any> {
    fun myFun(arg1: T, arg2: T){
        println(arg1.equals(arg2))
    }
}

fun main(args: Array<String>) {
    val obj = MyClass<String>()
    obj.myFun("hello", "hello")

    val obj2 = MyClass<Int?>()//error
    obj2.myFun(null, 10)
}
```

17.3. Variance

17.3.1. Variance란?

- 제네릭에서 Variance(가변, 공변)란 상하위 관계에서 타입 변형과 관련.




```
open class Super {
    open fun sayHello() {
        println("i am super sayHello...")
    }
}
class Sub: Super(){
    override fun sayHello() {
        println("i am sub sayHello....")
    }
}
fun main(args: Array<String>) {
    val obj: Super = Sub()
    obj.sayHello()

    val obj2: Sub = obj as Sub
    obj2.sayHello()
}
```

17.3. Variance

- 제네릭은 타입이지 클래스가 아니다. : invariance


```
open class Super

class Sub: Super()

class MyClass<T>

fun main(args: Array<String>) {
    val obj = MyClass<Sub>()

    val obj2: MyClass<Super> =  obj//error
}
```

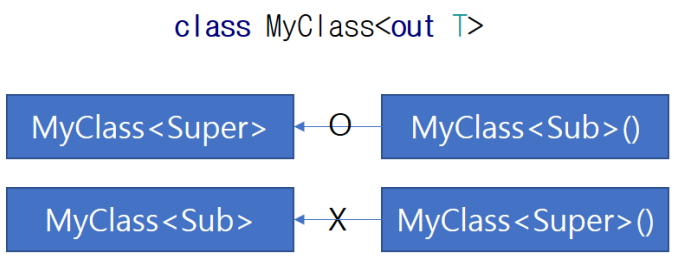
- MyClass<Sub>로 선언된 객체를 MyClass<Super>에 대입하기 위해서는 Variance 가 필요.
- out 과 in 어노테이션 이용

17.3. Variance

17.3.2. covariance

- out 어노테이션을 이용하여 하위 타입으로 선언된 객체를 상위 타입에 대입.

```
open class Super

class Sub: Super()

class MyClass<out T>

fun main(args: Array<String>) {
    val obj = MyClass<Sub>()
    val obj2: MyClass<Super> =  obj

    val obj3 = MyClass<Super>()
    val obj4: MyClass<Sub> = obj3//error
}
```



17.3. Variance

out 어노테이션을 사용하는 규칙

- 하위 제네릭 타입이 상위 제네릭 타입에 대입 가능
- 상위 제네릭 타입이 하위 제네릭 타입에 대입 불가능
- 함수의 리턴 타입으로 선언가능
- 함수의 매개변수 타입으로 선언 불가능
- val 프로퍼티에 선언가능
- var 프로퍼티에 선언 불가능

```
open class Super

class Sub: Super()

class MyClass<out T>(val data: T) {
    val myVal: T? = null
    var myVal2: T? = null//error
    fun myFun(): T {
        return data
    }
    fun myFun3(arg: T) { }//error
}
fun main(args: Array<String>) {
    val obj = MyClass<Sub>(Sub())
    val obj2: MyClass<Super> =  obj

    val obj3 = MyClass<Super>(Super())
    val obj4: MyClass<Sub> = obj3//error
}
```

17.3. Variance

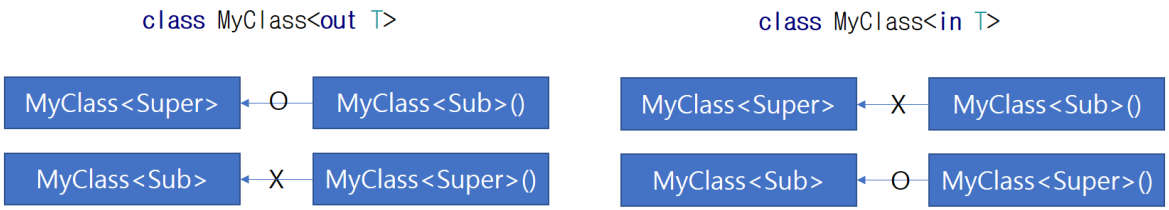
- `public interface MutableList<E> : List<E>, MutableCollection<E> { }`
- `public interface List<out E> : Collection<E> { }`

```
fun main(args: Array<String>) {
    val mutableList: MutableList<Int> = mutableListOf(10, 20)
    val mutableList2: MutableList<Number> = mutableList//error

    val immutableList: List<Int> = listOf(10, 20)
    val immutableList2: List<Number> = immutableList
}
```

17.3. Variance

17.3.3. contravariance

- in 어노테이션을 이용해 상위 제네릭 타입이 하위 제네릭 타입에 대입되어 사용.



in 어노테이션 규칙

- 하위 제네릭 타입이 상위 제네릭 타입에 대입 불가능
- 상위 제네릭 타입이 하위 제네릭 타입에 대입 가능
- 함수의 리턴 타입으로 선언 불가능
- 함수의 매개변수 타입으로 선언 가능
- val 프로퍼티에 선언 불가능
- var 프로퍼티에 선언 불가능

17.3. Variance

```
open class Super

class Sub: Super()

class MyClass<in T>() {

    val myVal: T? = null//error
    var myVal2: T? = null//error

    fun myFun(): T? {//error
        return null
    }
    fun myFun3(arg: T) { }
}

fun main(args: Array<String>) {
    val obj = MyClass<Sub>()
    val obj2: MyClass<Super> =  obj//error

    val obj3 = MyClass<Super>()
    val obj4: MyClass<Sub> = obj3
}
```

17.4. 타입 프로젝션

17.4.1. 이용측 Variance

- Declaration-Side Variance(선언위치 Variance) 와 Use-Side Variance(사용위치 Variance)
- in 과 out 을 사용하는 위치에 따른 구분

- class MyClass<out T>(val data: T) { } 선언위치 Variance.
- invariance로 선언된 클래스를 이용하는 곳에서 in, out 을 추가해서 variance가 가능하게 하는 것이 사용위치 Variance

```
class MyClass<T>(val data: T){
    fun myFun(): T {
        return data
    }
    fun myFun2(arg: T){   }
    fun myFun3(arg: T): T{
        return data;
    }
}
fun some1(arg: MyClass<in Int>){
    arg.myFun()
    arg.myFun2(10)
    arg.myFun3(10)
}
fun main(args: Array<String>) {
    some1(MyClass<Int>(10))
    some1(MyClass<Number>(10))
}
```

17.4. 타입 프로젝션

- 사용위치 out 어노테이션

```
class MyClass<T>(val data: T){
    fun myFun(): T {
        return data
    }
    fun myFun2(arg: T){

    }
    fun myFun3(arg: T): T{
        return data;
    }
}

fun some2(arg: MyClass<out Number>){
    arg.myFun()
//    arg.myFun2(10)//error
//    arg.myFun3(10)//error
}

fun main(args: Array<String>) {
    some2(MyClass<Number>(10))
    some2(MyClass<Int>(10))
}
```

17.4. 타입 프로젝션

- 사용위치 Variable 사례

```
public class Array<T> { }
```

invariane로 이용

```
fun copy(from: Array<Int>, to: Array<Int>) {
    for (i in from.indices)
        to[i] = from[i]
}

fun main(args: Array<String>) {
    val array1: Array<Int> = arrayOf(1, 2, 3)
    val array2 = Array<Int>(3){ x -> 0}
    copy(array1, array2)
    array2.forEach { println(it) }
}
```

```
fun copy(from: Array<Any>, to: Array<Any>) {
    for (i in from.indices)
        to[i] = from[i]
}
fun main(args: Array<String>) {
    val array1: Array<Int> = arrayOf(1, 2, 3)
    val array2 = Array<Any>(3){ x -> 0}
    copy(array1, array2)//error
    array2.forEach { println(it) }
}
```

17.4. 타입 프로젝션

```
fun copy(from: Array<out Any>, to: Array<Any>) {
    for (i in from.indices)
        to[i] = from[i]
}

fun main(args: Array<String>) {
    val array1: Array<Int> = arrayOf(1, 2, 3)
    val array2 = Array<Any>(3){ x -> 0}
    copy(array1, array2)//error
    array2.forEach { println(it) }
}
```

17.4. 타입 프로젝션

17.4.2. 스파(*) 프로젝션

- 스타 프로젝션이란 제네릭 타입을 <*> 로 표현하는 것을 의미
- 선언위치에서는 불가능하며 사용위치에서만 허용

```
//class MyClass<*>//error

class MyClass2<T>

fun myFun(arg: MyClass2<*>){ }
```

- 스타 프로젝션은 제네릭 타입을 모른다는 의미
- <Any?>는 정확한 타입이 명시된 것이고 <*>은 타입을 모른다는 것의 차이

```
val list2: MutableList<Any?> = mutableListOf(10, 10.0, "kkang")
list2.forEach{ println(it)}

val list3: MutableList<*> = mutableListOf(10, 10.0, "kkang")
list2.forEach{ println(it)}
```

17.4. 타입 프로젝션

```
val list2: MutableList<Any?> = mutableListOf<Any>(10, 10.0, "kkang")//error
list2.forEach{ println(it)}

val list3: MutableList<*> = mutableListOf<Any>(10, 10.0, "kkang")
list2.forEach{ println(it)}
```

- 목적은 다르지만 의미상으로 봤을때는 <*>은 <out Any?>와 동일.

```
fun some(array: MutableList<Int>){
    array.add(10)
}
fun some1(array: MutableList<out Any?>){
    array.add(10)//error
}
fun some2(array: MutableList<*>){
    array.add(10)//error
}
fun main(args: Array<String>) {

    val list1 = mutableListOf<Int>(10, 20)
    some1(list1)

    val list2 = mutableListOf<Int>(10, 20)
    some2(list2)
}
```

17.4. 타입 프로젝션

- 선언위치에 제네릭 타입이 <in T> 로 선언된 경우 이를 이용할 때 <*> 으로 사용하는 것은 <in Nothing>으로 이용되는 것과 동일

```
class MyClass<in T>{
    fun myFun(a: T){ }
    fun myFun2(){}
}

fun some(arg: MyClass<*>){
    arg.myFun(10)//error
    arg.myFun2()
}
fun some1(arg: MyClass<in Any?>){
    arg.myFun(10)
    arg.myFun2()
}
fun some2(arg: MyClass<in Nothing>){
    arg.myFun(10) //error
    arg.myFun2()
}
```

17.5. 실행 시점의 제네릭

17.5.1. 제네릭과 as, is 이용

- 제네릭 정보는 컴파일러를 위한 정보

- 제네릭 정보가 컴파일 될 때 사라지게 됨으로서 as 혹은 is 사용에 주의

```
fun some(arg: Any){
    if(arg is Int){

    }
    val intVal=arg as Int
    intVal.plus(10)
}

fun main(args: Array<String>) {
    some(10)
    some("hello")
}
```

17.5. 실행 시점의 제네릭

- "Cannot check instance of erased type" 이라는 컴파일 에러

```
fun some1(arg: List<Int>){
    if(arg is List<Int>){
        println(arg.sum())
    }
}

fun some2(arg: List<*>){
    if(arg is List<Int>){//error

    }
}
```

17.5. 실행 시점의 제네릭

- as 이용의 경우

```
fun some3(arg: List<*>){
    val intList = arg as List<Int>
    println(intList.sum())
}

fun main(args: Array<String>) {
    some3(listOf(10, 20))
    some3(listOf("hello", "kkang"))
}
```

- "Unchecked cast: List<*> to List<Int>" 라는 경고
- 런타임 시점에 ClassCastException 이 발생

17.5. 실행 시점의 제네릭

17.5.2. 인라인 함수와 reified 이용

- 제네릭 타입을 실행시점에 알아내서 as 와 is가 정상적으로 동작하게 할 수 있는 방법
- reified 을 이용해 형식 타입이 선언되면 이 제레릭 타입은 시행시점까지 유지
- reified는 inline 함수내에서만 사용이 가능

```
inline fun <reified T> some(arg: Any){
    if(arg is T){
        println("true")
    }else {
        println("false")
    }
}

fun main(args: Array<String>) {
    some<String>("hello")
    some<Int>("hello")
}
```

17.6. Unit, Nothing 타입

17.6.1. Unit 타입

자바의 void와 Unit의 차이

```
public class JavaTest {
    public void javaFun(){ }

    public static void main(String[] args){
        JavaTest obj=new JavaTest();
        System.out.println(obj.javaFun());//error
    }
}
```

```
fun myFun1(){ }
fun myFun2(): Unit { }
```

17.6. Unit, Nothing 타입

- void는 함수의 리턴값이 없다는 일종의 예약어이지만 Unit은 타입

```
fun myFun1(){ }
fun myFun2(): Unit { }

fun myFun3(): Unit {
    return Unit
}

val myVal: Unit = Unit

fun main(args: Array<String>) {
    println(myFun1())
}
```

실행결과

```
kotlin.Unit
```

- Unit은 kotlin.Unit만 대입되는 특이한 타입

17.6. Unit, Nothing 타입

- 제네릭에서 Unit의 이용

```
interface MyInterface<T> {
    fun myFun(): T
}

class MyClass: MyInterface<String> {
    override fun myFun(): String {
        return "hello"
    }
}

class MyClass2: MyInterface<Unit> {
    override fun myFun() {

    }
}
```

17.6. Unit, Nothing 타입

17.6.2. Nothing

- Nothing 타입으로 선언이 되면 이곳에는 null 만 대입
- Nothing은 결국 값이 없다는 것을 명시적으로 표현하기 위해서 사용

```
fun myFun(arg: Nothing?): Nothing {
    throw Exception()
}
val myVal: Nothing? = null
```

함수의 리턴 타입으로 Nothing 사용

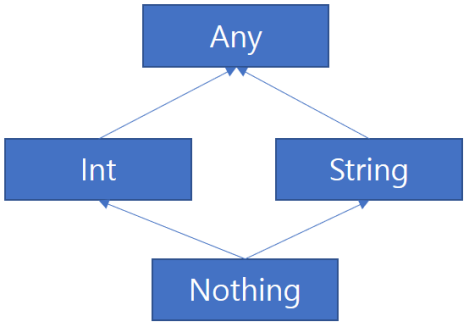
- 함수는 리턴이 없다는 것을 명시적으로 선언하고 싶은 경우

```
fun myFun(): Nothing {
    while(true){
        //.....
    }
}
fun myFun2(): Nothing? {
    return null
}
fun myFun3(): Nothing {
    throw Exception()
}
```

17.6. Unit, Nothing 타입

제네릭에서 Nothing의 이용

- Nothing 타입은 다른 어떤 타입의 프로퍼티에도 대입이 가능




```
val myVal1: Nothing? = null

val myVal2: Int? = myVal1
val myVal3: String? = myVal1
```

17.6. Unit, Nothing 타입

```
class MyClass<T>

fun someFun(arg: MyClass<in Nothing>){ }

fun main(args: Array<String>) {
    someFun(MyClass<Int>())
    someFun(MyClass<String>())
}
```